Membership Protocols for Asymmetric Groups[*]

Karlo Berket and Deborah A. Agarwal
Ernest Orlando Lawrence Berkeley National Laboratory
1 Cyclotron Rd, MS 50B-2239
Berkeley, CA 94720
phone: (510) 486-4807 fax: (510) 486-6363
{KBerket, DAAgarwal}@lbl.gov

P. Michael Melliar-Smith and Louise E. Moser
University of California, Santa Barbara
Electrical and Computer Engineering Department
Santa Barbara, CA 93106
{pmms, moser}@ece.ucsb.edu

Abstract

Many group communication systems have been built to operate on local-area networks. One of the principal problems with scaling these systems has been the need to maintain a consensus-based membership for the entire group. The membership is a necessary component to providing such properties as group ordering and virtual synchrony. This paper presents a novel approach to addressing this problem. It recognizes and leverages off the fact that the processes in a group are not necessarily equal. This allows us to form an asymmetric membership with the consensus-based membership containing a subset of the group. We have developed membership protocols for use in such groups and present them in this paper as a solution to the scalability problem of consensus-based membership protocols.

1. Introduction

Reliable group ordered delivery of multicast messages in a distributed system consisting of multiple sender is a useful service that simplifies the programming of distributed applications. With such a delivery service, provided by group communication protocols, all processes in each group of the application receive the same messages in the same order. Such a service helps to maintain the consistency of replicated information and to coordinate the activities of the various processes.

[*]This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division, U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

In addition, group communication protocols provide a membership service that allows the system to make progress in the presence of process faults. Such protocols require the group membership and delivery order to obey a form of virtual synchrony [6, 8, 13].[1] Virtual synchrony and its variations define consistency constraints on processes transitioning between memberships. Messages are delivered within the context of views, *i.e.,* every message delivered to the application has a view associated with it. A view has an associated group membership, and the membership mechanisms are in charge of installing view changes. Typical applications that might use a group communication system include state-machine replication ([9, 10, 14]), distributed transactions and database replication ([16]), load balancing ([11]), system management ([4]), system monitoring ([3]), highly available servers ([15]), and collaborative computing ([1, 2]).

With the increasing popularity of the Internet, there is an increasing interest in group communication protocols that are scalable to the environment of the Internet. IP multicast provides a scalable best-effort multicast service for the Internet, and is a valuable building block for group communication systems.

The main hindrances to scalability of existing group communication systems are the message delivery guarantees provided to the application and their effect on the way in which membership is maintained. Group communication protocols require the group membership and the delivery order to obey a form of virtual synchrony.

Such requirements result in expensive (in time and messages required) membership repair algorithms. There appears to be no way around those membership repair algorithms, which require a consensus decision to be made. The message cost of those algorithms is $O(n^2)$, where n is the number of processes in the group. Furthermore, the interval between membership changes is inversely proportional to n and, thus, if the value of n is large, too much time can be spent in the membership protocol itself.

The Internet environment adds to these costs. It is more prone to errors, causing the interval between membership changes to decrease more rapidly with an increase in the number of processes in the group. The time cost for the membership repair algorithms is dependent on the round-trip time between the two furthest processes in the group. Thus, in a wide-area environment, such as the Internet more time is spent in the membership protocols themselves.

Also, to ensure virtual synchrony, many group communication protocols stop delivering messages while membership changes are taking place and, thus, for large values of n, they might deliver no messages at all. Nevertheless, to obtain a consistent view of the membership and to ensure the message delivery guarantees, membership repair algorithms must be run.

In traditional group communication systems, every process in a process group is treated as an equal. When the system requires a consensus decision, such as that used in a membership repair algorithm, every process participates. This is because the membership protocols of these systems attempt to have every process in the process group in the membership set. Every process in these systems executes the same membership algorithms. Thus, we refer to these groups as *symmetric groups*.

In the InterGroup system, we concentrate on providing the reliable group ordered delivery service. To accomplish this we only need to explicitly track the membership of the processes that are sending application data. In our experience, most applications that use group communication have only a minority of the processes sending application data at any given time. Thus, we cut down on the cost of the membership repair algorithms using the following strategy.

[1]Some protocols require that, for each message delivered, a membership regarding that message be delivered. This results in running a consensus algorithm for every message, resulting in severe scalability problems, so these protocols will not be discussed.

In the InterGroup system, not all processes are equal. In each process group, a process is classified by its recent activity. If the process has been sending data to the group recently, it is classified as an *active sender*. Active senders are members of the *sender group* of the process group. Only the senders in the process group need to participate in the majority of consensus decisions. Thus, the membership algorithms that are executed at a process depend on the state that the process is in. We refer to these groups as *asymmetric*.

We also use voluntary mechanisms for entering and leaving the group. Voluntary mechanisms can take advantage of the properties of the system to avoid executing the membership repair algorithms.

We first describe the model of our system. Then we discuss the data structures and messages used in our algorithms. This is followed by a description of the process group membership protocol. Then we present the membership repair algorithm, followed by the receiver membership repair algorithm. Finally we conclude and suggest future work.

2. Model

We consider an asynchronous distributed system consisting of n processes $p_1, p_2, \ldots, p_n$ that communicate via messages over a network. Each process within the system has a unique identifier. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Processes have access to Lamport clocks.[12]

The system consists of m *process groups* $g_1, g_2, \ldots, g_m$. Each process group g_i consists of l processes $q_1, q_2, \ldots, q_l$ that communicate via messages over a network. Each process within the system may belong to multiple process groups.

We assume the existence of a *control hierarchy*. This control hierarchy provides a means to aggregate data from all of the processes in a process group and return a single value based on a simple function (*e.g.,* maximum value). We have developed such a control hierarchy and it is described in [7].

The rest of the discussion is restricted to a single process group.

A view of a process group is uniquely identified by its *membership*, and *view identifier*. The membership consists of a set of process identifiers. The view identifier consists of three fields:

- *timeEntered*: the logical time at which this view begins
- *leader*: the identifier of a process chosen deterministically from the membership
- *leaderSeq*: the leader sequence number

The *current view* at a process is the most recent view of the process group provided to the application at that process. The goal is for all processes to have the same current view..

The process group comprises two disjoint groups: the *sender group*, and the *receiver group* which consists of all the other processes in the process group. Each process in the process group is either in the sender group or the receiver group at any given time (except during startup). If a process is in the membership of its current view, it is in the sender group.

A process that wishes to send messages within the group joins the sender group. Multicast messages are assumed to be delivered to the membership protocols, reliably and in source order.

The network is allowed to partition and remerge.

A process and network fault detector is present. The process and network fault detector is responsible for determining process failures and the reachability of processes, and is based on timeouts. The process and network fault detector analyzes every message received from the process group. If it does not receive a message from a process for a predetermined amount of time it notifies the system of this via a PFailure message. In order for this approach to work, processes must send messages periodically even when they have no data to send. This requirement is satisfied by having every process that is sending data also periodically send Keep Alive messages.

3. The Data Structures

The process group membership protocol employs a number of message types to ensure that a valid membership view is executing. In this section, we introduce the messages that are internal to a process, the data message type whose fields are included in all reliable messages sent over the network, the network messages used in the membership protocols, and the internal data structures.

3.1. Internal Messages (Events)

Process Failure Message
Process Failure (PFailure) messages are used to signal the suspicion that a process in the current view or proposed membership has failed. A PFailure message contains the following fields:

- *type*: PFailure
- *proc*: The process identifier of the process that is suspected to have failed.

Process Foreign Message
Process Foreign (PForeign) messages are used to signal the reception of a message from a process that is not in the membership of the current view. A PForeign message contains the following fields:

- *type*: PForeign
- *proc*: The process identifier of the process that is suspected not to be in the membership of the current view.

To Sender Message
To Sender (ToSender) messages are used to signal that this process has decided to enter the sender group. A ToSender message contains the following fields:

- *type*: ToSender

To Receiver Message
To Receiver (ToRec) messages are used to signal that this process has decided to exit the sender group. A ToRec message contains the following fields:

- *type*: ToReceiver

User Membership Change Message
User Membership Change (UMC) messages are used to signal a view change to the application. They are delivered to the application at the point in the data stream when the view change occurs. A UMC message contains the following fields:

- *type*: UMC
- *membID*: The unique identifier of the view whose beginning this message signals.
- *cut*: The time at which this view begins.
- *memb*: The set of processes in the membership of the view whose beginning this message signals.
- *transSet*: The set of processes that were in the membership of the previous view, and are in the membership of the view whose beginning this message signals.

3.2. Data Messages

Data messages are the basis of all network messages. The following fields are contained in a Data message header:

- *type*: The message type of this message.
- *sender*: The unique identifier of the process that sent this message.
- *seq*: The sequence number of this message. If another message from this sender has the same value of this field, that message must be identical to this one or a Keep Alive message. This field is used to detect message loss and allow FIFO ordering for messages from individual processes.
- *timestamp*: The timestamp of this message. This field is used to determine a process group wide ordering of messages.

3.3. Network Messages

Membership Change Message

This message extends the Data message. Membership Change (MC) messages signal that a new view should be installed. A MC message contains the following field which is a constant:

- *type* = MC

The following fields are added for a MC message:

- *membID*: The unique identifier of the new view.
- *procs*: The set of all of the identifiers of the processes that are in the membership of the new view.
- *oldIDs*: A set of identifiers of all of the latest views of all processes in *procs*.
- *cutInfo*: An array indexed by the view identifiers in *oldIDs*. The values in the array are the sequence numbers of the last message delivered in each of these views.
- *lastSeq*: An array indexed by the process identifiers in *procs*. The values in the array are the sequence numbers of the first message to be delivered, in the next view, from each of these processes.

Process Add Message
This message extends the Data message. Process Add (PAdd) messages are used for voluntary joins to the sender group. They are sent by a process in the sender group, on behalf of a process that wishes to join the sender group. A PAdd message contains the following field which is a constant:

- *type* = PAdd

The following field is added for a PAdd message:

- *proc*: The unique identifier of the process that wishes to join the sender group.

Process Leave Message
This message extends the Data message. Process Leave (PLeave) messages are used for voluntary leaves from the sender group. A process that wishes to leave the sender group, creates this message and sends it to the process group. A PLeave message contains the following field which is a constant:

- *type* = PLeave

Process Join Message
This message extends the Data message. Process Join (PJoin) messages are used to reach consensus on the processes that will be in the membership of the next view (*proposed membership*). A PJoin message contains the following field which is a constant:

- *type* = PJoin

The following fields are added for a PJoin message:

- *membID*: The unique identifier of the current view at the message sender.
- *procs*: A set of identifiers of the processes that, at the time this message was sent, were being considered for the membership of the next view at the message sender.
- *fail*: A set of identifiers of the processes that, at the time this message was sent, were considered as having failed. This set is a subset of procs.

Process Failure Block Message
This message extends the Data message. Process Failure Block (PFailBlock) messages are used to signal a suspicion that a process in the current view or proposed membership has failed. These messages are generated by a process that is not able to receive a retransmission of a message for a specified period of time. A PFailBlock message contains the following field which is a constant:

- *type* = PFailBlock

The following fields are added for a PFailBlock message:

- *proc*: The unique identifier of the process that is suspected to have failed.
- *procSeq*: The sequence number of the message sent by *proc* that has not been received.

Recovery Information Message
This message extends the Data message. Recovery Information (RecInfo) messages are used to distribute the information regarding the time when a view change should occur. A RecInfo message contains the following field which is a constant:

- *type* = RecInfo

The following fields are added for a RecInfo message:

- *membID*: The unique identifier of the sender's current view.
- *cut*: The timestamp of the last message delivered to the user.
- *seqCut*: An array referenced by the unique identifier of the processes in the membership of the sender's current view that have been tagged as being suspected of failing. The values in the array are the sequence numbers of the last message that the sender of this message considers can be delivered in reliable source order from each of these processes.

Ready To Commit Message
This message extends the Data message. Ready To Commit (RTC) messages are used to signal that the sending process is ready to install the proposed membership. A RTC message contains the following field which is a constant:

- *type* = RTC

The following field is added for a RTC message:

- *firstSeq*: The sequence number of the first message to be delivered from this process in the next view, if it is installed.

3.4. Internal Variables

The membership protocols use the following internal variables:

- *state*: The state of the membership protocol. These states are described in Section 4.
- *blockSet*: The set of processes that are blocked from consideration for further memberships. Processes are added to this set during an execution of the MRA. Processes are removed from this set after a certain amount of time (this amount of time should be set so that short-term fluctuations do not effect the membership, we use a value that is 100 times the expected latency between the most distant processes in the group), though they may not be removed by a process in the *MRA* state.

4. The Process Group Membership Protocol

The process group membership protocol is used to keep the processes in the process group executing consistently and with consistent views of the membership boundaries, despite changes in the membership (voluntary or not). We describe here the states of the membership protocol and the processing involved to keep the process group executing. There are two states in the membership protocol that may be considered as normal or stable. These are the *Receiver* state and the *Sender* state. The process will always tend to one of these states.

First, we describe how the membership protocol is initialized. After that, we describe what occurs when a process is forced out of one of the stable states. The protocols used in the case of faults (the repair algorithms) are described in detail in Sections 5 and 6.

4.1. Bootstrap

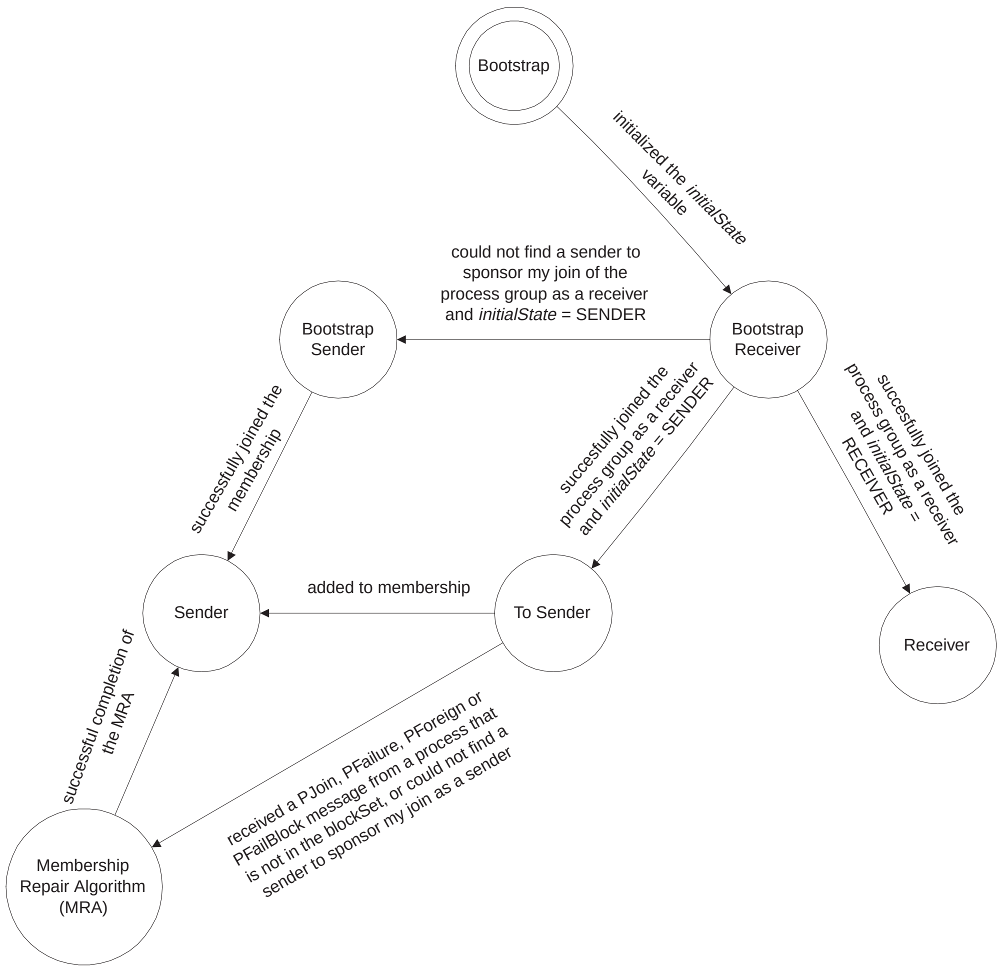
Bootstrap
initialized the *initialState* variable
could not find a sender to sponsor my join of the process group as a receiver and *initialState* = SENDER
Bootstrap Sender
Bootstrap Receiver
successfully joined the membership
succesfully joined the process group as a receiver and *initialState* = SENDER
succesfully joined the process group as a receiver and *initialState* = RECEIVER
Sender
added to membership
To Sender
Receiver
successful completion of the MRA
received a PJoin, PFailure, PForeign or PFailBlock message from a process that is not in the blockSet, or could not find a sender to sponsor my join as a sender
Membership Repair Algorithm (MRA)

Figure 1. State machine for the initialization of the membership protocols.

The state machine for the initialization or bootstrap of the membership protocols is shown in Figure 1. When a process wishes to join a process group, it must do so using the membership protocol, entering through the *Bootstrap* state. A joining process may specify whether it wishes to join the sender group directly, or whether it wishes to join the process group as a receiver. This is specified by an argument to the membership protocols and stored in the variable *initialState*. The allowed values for *intialState* are SENDER (in case the process wishes to join the sender group directly) and RECEIVER (otherwise).

The *Bootstrap Receiver* state is entered after the *initialState* variable is initialized. A process's objective in this state is to obtain enough information about a current view, so that this process (process *q*) can join the process group as a receiver. To obtain this information, process *q* must first find a process that is a member of a sender group in this process group. This step is accomplished by receiving messages addressed to the process group and attempting to contact the senders of those messages. Once a process in the sender group is contacted, the necessary information is obtained from that process (process *p*). This information consists of:

- The unique identifier of the current view at process *p*.
- The identifiers of the processes in the membership of the current view at process *p*.
- The *sequence cut* which consists of a sequence number for each process in the membership of the current view at process *p*. Process *q* will begin reliable delivery of messages for each of these processes with the message that has a sequence number matching this sequence number.
- The timestamp at which process *q* will install this view, *i.e.,* process *q* will deliver messages, after this timestamp and will have the same current view as process *p*. This timestamp must be greater than or equal to the highest timestamp of any message that is represented by the sequence cut.

Once process *q* installs this view, it enters the *Receiver* state if *initialState* is RECEIVER and finishes the bootstrap process. If process *q* installs this view and *initialState* is SENDER, it enters the *To Sender* state. If process *q* is not able to contact a member of the sender group and *initialState* is SENDER, it enters the *Bootstrap Sender* state. This usually occurs when there are no processes currently in the sender group.

When a process that is in the receiver group, wishes to join the sender group, it enters the *To Sender* state. First, the process tries to find a process in the sender group of its current view that will sponsor its join of the sender group. If it cannot find one, it transitions to the *MRA* state, described in Section 5. If this process does find such a process, called a *sponsor*, this process requests its sponsor to send a PAdd message with this process' unique identifier as the *proc* field of that message. Once this request is confirmed, this process waits for that PAdd message to be delivered to the user. If the request is not confirmed, this process looks for another sponsor. If while it awaits delivery of the PAdd message, one of the following events occurs:

- Arrival of a PJoin message from a process that is not in the *blockSet*,
- Arrival of a PFailure message about a process that is in the membership of its current view,.
- Arrival of a PForeign message regarding a process that is not in the *blockSet*, and
- Arrival of a PFailBlock message from a process that is in the membership of its current view

this process transitions to the *MRA* state. Once the PAdd message adding this process to the sender group (actually a UMC message denoting the start of a new view that includes this process in the membership) is delivered, this process transitions to the *Sender* state and the bootstrap is finished.

The *Bootstrap Sender* state is used to add this process to the sender group, if *initialState* is SENDER and this process couldn't successfully join the process group as a receiver first. The processing inside this state is the same as in the *MRA* state, however, the sending of user messages is disabled until the repair algorithm is concluded. Upon the successful completion of the repair algorithm, the process transitions to the *Sender* state and enables the sending of user messages. Thus, completing the bootstrap.

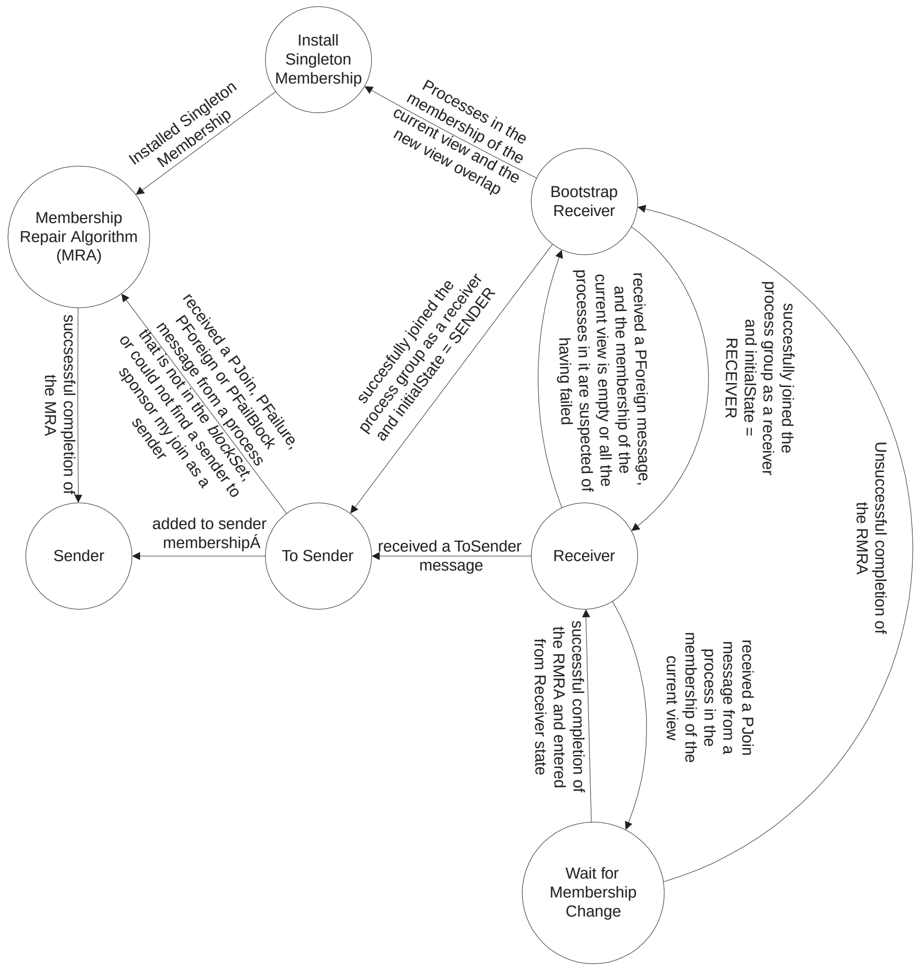
Install Singleton Membership
Installed Singleton Membership
Processes in the membership of the current view and the new view overlap
Membership Repair Algorithm (MRA)
Bootstrap Receiver
successful completion of the MRA
received a PJoin, PFailure, PForeign or PFailBlock message from a process that is not in the blockSet, or could not find a sender to sponsor my join as a sender
succesfully joined the process group as a receiver and initialState = SENDER
received a PForeign message, and the membership of the current view is empty or all the processes in it are suspected of having failed
succesfully joined the process group as a receiver and initialState = RECEIVER
Unsuccessful completion of the RMRA
Sender
added to sender membershipÁ
To Sender
received a ToSender message
Receiver
successful completion of the RMRA and entered from Receiver state
received a PJoin message from a process in the membership of the current view
Wait for Membership Change

Figure 2. How the *Receiver* state is left in the membership protocols.

4.2. Leaving the Receiver State

The state diagram showing what happens when a process leaves the *Receiver* state is shown in Figure 2. When a process is in the *Receiver* state, it is in a normal or stable state. The process executes as if the membership of the process group was static. There are only four reasons for the process to leave this state: (1) it wishes to join the sender group, (2) a process in the membership of its current view has started a membership repair algorithm, (3) this process suspects that all of the processes in the sender group have failed and it has received a message from a process that has a different view, or (4) the membership is empty in its current view and this process has received a message from a process that has a different view. We describe the results of these four scenarios below, starting from the last and moving forward.

A view with an empty membership can be installed only if the last process in the membership leaves the sender group voluntarily. At this point, the process in the *Receiver* state has nothing to do. If the process receives a PForeign message while the sender group is empty, it behaves as if it is initializing the membership protocol with *initialState* set to RECEIVER. Thus, it transitions to the *Bootstrap Receiver* state and follows the transitions described previously.

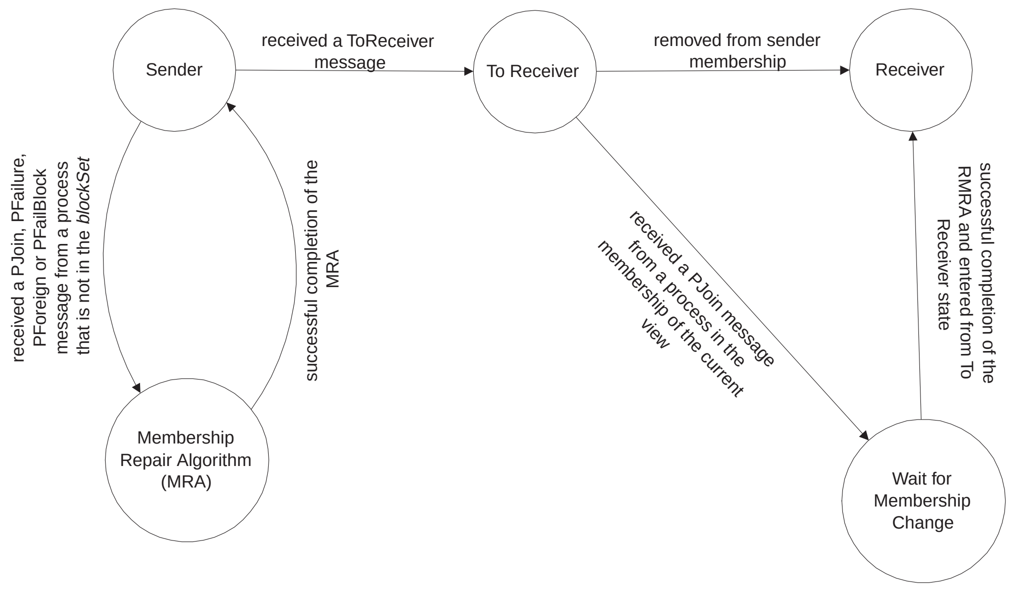
Sender
received a ToReceiver message
To Receiver
removed from sender membership
Receiver
received a PJoin, PFailure, PForeign or PFailBlock message from a process that is not in the *blockSet*
successful completion of the MRA
Membership Repair Algorithm (MRA)
received a PJoin message from a process in the membership of the current view
successful completion of the RMRA and entered from To Receiver state
Wait for Membership Change

Figure 3. How the *Sender* state is left in the membership protocols.

If this process suspects that all of the processes in the sender group have failed, it behaves as if the sender group were empty, and switches to the *Bootstrap Receiver* state. However, if any of the processes that were in the membership of its view appear in the membership of the view it receives while it is in the *Bootstrap Receiver* state, it follows a new transition. In this case it transitions to the *Install Singleton Membership* state, where it installs a view whose membership consists of only itself. Once that is accomplished, it transitions to the *MRA* state.

When a process in the sender group starts a membership repair algorithm, it sends a PJoin message as part of that algorithm. Thus, on reception of a PJoin message from a process in the membership of the current view, this process, in the *Receiver* state, transitions to the *Wait for Membership Change* state, described in Section 6.

When a process in the *Receiver* state wishes to join the sender group, it switches to the *To Sender* state. The *To Sender* state and the transitions from it were described in Section 4.1.

4.3. Leaving the Sender State

The state diagram that indicates what happens when a process leaves the *Sender* state is shown in Figure 3. When a process is in the *Sender* state, it is in a normal or stable state. The process executes as if the membership of the process group were static. There are only three reasons for the process to leave this state: (1) it wishes to leave the sender group, (2) a process in the sender group is suspected of having failed, or (3) it receives a message from a process that is not in the membership of its current view. We describe these three scenarios below.

If a process that is in the *Sender* state wishes to leave the sender group, it sends a PLeave message to the process group, stops sending new messages to the group, and transitions to the *To Receiver* state. Once in the *To Receiver* state, this process waits for the PLeave message it sent to be delivered to the user. When this message is delivered, the process transitions to the *Receiver* state. However, if the process receives a PJoin message from a process in the membership of its current view before it receives the PLeave message, then the process transitions to the *Wait for Membership Change* state, described in Section 6.

If a process suspects a member of its current view of having failed, one of the following events occurs:

- The fault detector of this process generates a PFailure message concerning the process suspected to have failed, and the membership protocol receives that message.
- This process receives a PJoin message from a process in the membership of its current view.
- This process receives a PFailBlock message from a process in the membership of its current view.

The occurrence of any one of these events will cause the process to transition out of the *Sender* state into the *MRA* state, described in Section 5.

If the process receives a PForeign message regarding a process that is not in the membership of its current view, and is not in the *blockSet*, it transitions to the *MRA* state. This event signals that there is at least one more disjoint view installed at a different process in the process group. By switching to the *MRA* state and running the repair algorithm, the protocol attempts to merge these views into one.

5. Membership Repair Algorithm

The membership repair algorithm (MRA) is executed at a process that wishes to be in the membership of the next view. The MRA is run while the process is in the *Bootstrap Sender* state, described in Section 4.1, and in the *MRA* state of the process group membership protocol. The entry conditions of the *MRA* state have been discussed in Section 4. A process leaves the *MRA* state only after it successfully completes the MRA and then transitions to the *Sender* state.

We discuss the MRA in detail, because it allows the membership protocols to deal with failures of processes, partitioning of the process group, and merging of components of the partitions. We discuss the messages and internal variables first, followed by the protocol.

5.1. Data Structures

The MRA employs the following messages: PJoin, PFailure, PFailBlock, RecInfo, RTC, MC and UMC. All of these messages are described in Section 3. In addition to the *blockSet* variable common to the entire membership protocol, the MRA uses the following internal variables:

- *procSet*: The set of processes that are being considered for the next membership. This set can only grow during the run of an MRA.
- *failSet*: The set of processes that are being considered as failed for the next membership. This set is a subset of *procSet*. It can only grow during the execution of an MRA.
- *lastProcTable*: A mapping where the keys are processes in the setMinus (*procSet*, *failSet*) and the entries are sets containing the last known *procSet* for each of these processes. The sets are obtained from PJoin messages sent by the processes.
- *lastFailTable*: A mapping where the keys are processes in the setMinus (*procSet*, *failSet*) and the entries are sets containing the last known *failSet* for each of these processes. The sets are obtained from PJoin messages sent by the processes.

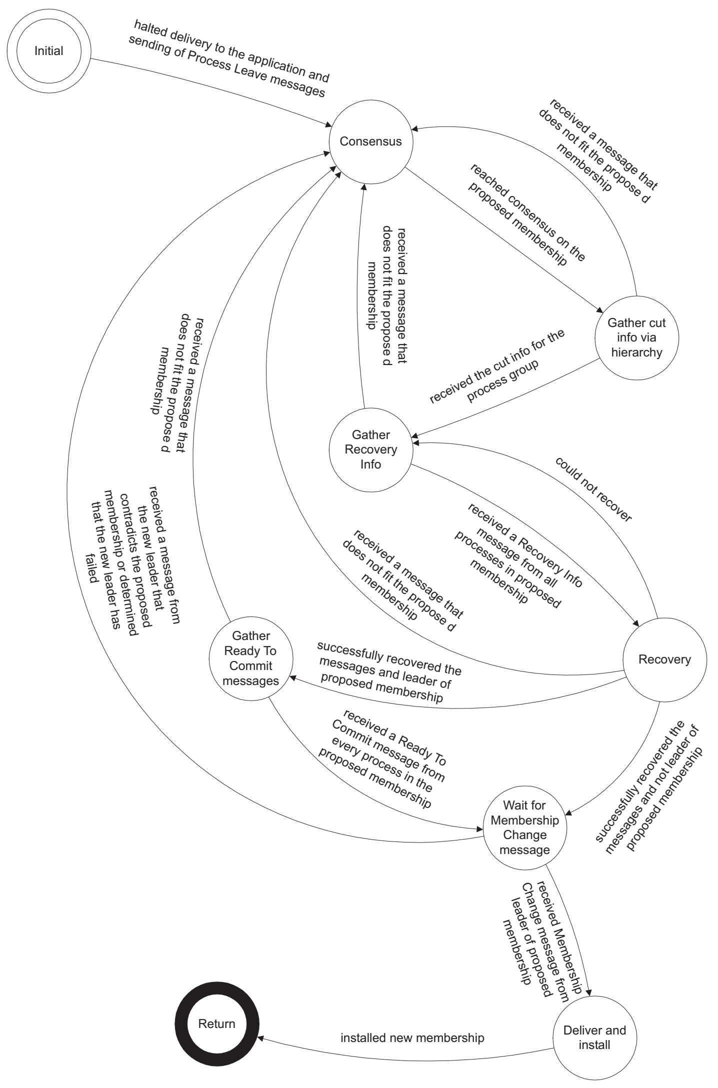
Initial
halted delivery to the application and sending of Process Leave messages
Consensus
received a message that does not fit the propose d membership
reached consensus on the proposed membership
Gather cut info via hierarchy
received a message that does not fit the propose d membership
received the cut info for the process group
received a message that does not fit the propose d membership
Gather Recovery Info
could not recover
received a message from the new leader that contradicts the proposed membership or determined that the new leader has failed
received a Recovery Info message from all processes in proposed membership
received a message that does not fit the propose d membership
Gather Ready To Commit messages
successfully recovered the messages and leader of proposed membership
Recovery
received a Ready To Commit message from every process in the proposed membership
successfully recovered the messages and not leader of proposed membership
Wait for Membership Change message
received Membership Change message from leader of proposed membership
Return
installed new membership
Deliver and install

Figure 4. The Membership Repair Algorithm

- *pendingDelivery*: A FIFO queue of messages that are ready to be delivered to the user.

5.2. The Protocol

The state machine for the MRA is shown in Figure 4. The process enters the MRA by entering the *Initial* state, where the MRA is initialized. The delivery of messages to the user is halted and the sending of PLeave messages is disallowed. The delivery of messages to the user needs to be suspended, because a *cut* (the place in the data stream delivered to the user) where the current view ends needs to be agreed upon in the membership repair algorithm. If message delivery were allowed, the cut that a process sends to the process group might not be valid. The sending of PLeave messages is disallowed, because it would cause the protocol to violate the termination property.

Upon the completion of these steps, the process transitions to the *Consensus* state. The operations performed in this state are based on the membership algorithms of the Transis system.[5] In the *Consensus*

The membership repair algorithm might "believe" that the message in question does not fit the proposed membership for one or more of the following reasons:

- The message is a PJoin message whose *procs* set is empty.
- The message is a PJoin message whose *sender* is not in the *procSet*.
- The message is a PJoin message whose *procs* set is not a subset of *procSet* and the *sender* is in the proposed membership.
- The message is a PJoin message whose *fail* set is not a subset of *failSet* and the *sender* is in the proposed membership.
- The message is a PForeign message whose *proc* identifier is not in the *procSet*.
- The message is a PFailure message whose *proc* identifier is in the proposed membership.
- The message is a PFailBlock message that this process has delivered and the *sender* of the PFailBlock message is in the proposed membership (the *sender* of this message is added to the *blockSet* as well).
- The message is a PFailBlock message that this process has not delivered and the process characterized by the *proc* field of the PFailBlock message is in the proposed membership is added to the *blockSet*.

Figure 5. Reasons for restarting consensus.

state the processes participating in the MRA attempt to reach agreement on the processes that will be in the membership of the next view. The exit condition for the *Consensus* state is that all of the processes that are continuing have agreed on the processes that are going to be in the membership of the next view, *i.e.,* the *proposed membership*. Once this condition is met, the process transitions to the *Gather Cut Info* state. More information on the *Consensus* state and the algorithm used to determine whether the exit condition has been met can be found in [7].

The process, upon entering the *Gather Cut Info* state, determines the local cut information for its current view, and sends it to the control hierarchy. The control hierarchy aggregates the cut information of all of the processes having the same view, and returns a single cut value for that view. This cut value represents the maximum information that can be recovered from this view. Upon receiving this information from the hierarchy, the process sends a RecInfo message, as a representation of this information, to the process group, and transitions to the *Gather Recovery Info* state. If the process, while in this state, receives a message that does not fit the proposed membership (for explanation, see Figure 5), it transitions back to the *Consensus* state.

Once in the *Gather Recovery Info* state, the process waits to receive a RecInfo message from every process in the proposed membership. The gathering of the cut information through the hierarchy may not provide consistent cut information across the processes. The gathering of RecInfo messages from all of the processes in the proposed membership, provides a final cut for the process group. It allows processes that enter from the same view to rule out some of the inconsistencies between the structure of the process group and the control hierarchy. It also allows processes that enter from different views to share cut information and allows the process to determine the time at which the next view should be installed. Upon reception of all of the RecInfo messages, the process transitions to the *Recovery* state. If the process, while in this state, receives a message that does not fit the proposed membership (for explanation see Figure 5), it transitions back to the *Consensus* state.

The *Recovery* state is the state in which the process attempts to acquire all of the messages in its current view, that were sent before the cut signaling the end of that view. Upon entry to this state, the process calculates the cut for its current view, and the time at which the next view is to be installed, from the information in the RecInfo messages. Using this calculated information, the protocol obtains and orders all of the messages necessary to install the next view and places them in the *pendingDelivery* queue. When all of these messages are ready to be delivered, the process sends a RTC message (with the sequence number of the first message to be delivered from this process in the next view). After the RTC message is sent, the process transitions to the *Gather Ready To Commit Messages* state if it is the leader of the next view, or to the *Wait for Membership Change Message* state otherwise. The *leader of a view* is the process that has the lowest unique process identifier in the membership of that view. Thus, each process can determine who is the leader of the next view locally, by the lowest process identifier in the proposed membership. If the process, while in the *Recovery* state receives a message that does not fit the proposed membership (for explanation, see Figure 5), the protocol must reset the state of the *pendingDelivery* queue to the state it was in when recovery started. Once this is done, the process transitions back to the *Consensus* state. If this process is unable to recover all of the messages due to a failure of a process that was the only one holding a copy of a message that needed to be recovered, a PFailBlock message will be sent accusing an already failed process of failing. This action will cause the process to determine a new cut, and send a RecInfo message based on that cut. Once the RecInfo message is sent, the process waits to reset the state of the *pendingDelivery* queue to the state it was in when recovery started, and then transitions back to the *Gather Recovery Info* state.

The leader of the next view waits to receive RTC messages from every process in the proposed membership while it is in the *Gather Ready To Commit Messages* state. Once it has received all of the RTC messages, it chooses a unique identifier for the next view, constructs a MC message with that identifier, sends it to the process group, and transitions to the *Wait for Membership Change Message* state. If the leader, while in this state, receives a message that does not fit the proposed membership (for explanation, see Figure 5), it must reset the state of the *pendingDelivery* queue to the state it was in when recovery started. Once that task is finished, the leader transitions back to the *Consensus* state.

A process in the *Wait for Membership Change Message* state waits for a MC message from the leader of the next view. Once it receives this message, the process transitions to the *Deliver and Install* state. If the process, while in this state, suspects that the leader of the next view has failed or receives a message from the leader of the next view that does not fit the proposed membership (for explanation, see Figure 5), it must reset the state of the *pendingDelivery* queue to the state it was in when recovery started. Once that task is finished, the process transitions back to the *Consensus* state.

In the *Deliver and Install* state, the process delivers all of the messages in the *pendingDelivery* queue to the application, followed by a UMC message based on the information in the MC message received from the new leader. The next sequence number that each process in the membership of the new view delivers is set, according to the information in the MC message, if that value is not already larger. Finally, the process installs the new view, and resumes delivery of messages to the user and the sending of PLeave messages, before it exits the MRA.

6. Receiver Membership Repair Algorithm

The receiver membership repair algorithm (RMRA) is executed at a process that is not in the membership of its current view, or that has no intention to be in the membership of the next view. A process executes the RMRA while it is in the *Waiting for Membership Change* state of the process group membership protocol. We have discussed the entry conditions of the *Waiting for Membership Change* state previously. The *Waiting for Membership Change* state may be exited after completion of the RMRA. If the RMRA is successful, the process transitions back to the state from which it entered the *Waiting for Membership Change* state. If the RMRA is unsuccessful, the process transitions to the *Bootstrap Receiver* state.

We discuss the RMRA in detail, because it allows the membership protocol to deal with failures of processes, partitioning of the process group, and merging of partitions. It differs from the MRA, because the process does not actively participate in the exchange of membership messages, other than reporting its local cut information to the control group. It can only observe messages and, thus, making the correct decision is even more important. For this reason, we have had to allow the process to leave the RMRA, deeming it unsuccessful.

6.1. Data Structures

The RMRA employs the following messages: PFailure, MC and UMC. These messages are described in Section 3.

6.2. The Protocol

The state machine for the RMRA is shown in Figure 6. The process enters the RMRA by entering the *Initial* state. This is where the RMRA is initialized. The delivery of messages to the user is halted.

Upon the completion of this task, the process determines the local cut information for its current view and sends it via the control hierarchy, and then transitions to the *Wait for Membership Change Message* state.

A process in the *Wait for Membership Change Message* waits for a MC message that contains the current view identifier in the *oldIDs* field. Once it receives this message, it enters the *Recovery* state.

The *Recovery* state is the state in which the process attempts to acquire all of the messages in its current view, that were sent before the cut signaling the end of that view. Upon entry to this state, the process calculates the cut for its current view, and the time at which the next view is to be installed, from the information in the MC message. If the process has delivered messages following the calculated cuts, it declares itself failed, informs the application of this fact and exits. Otherwise, it obtains and orders all of the messages necessary to install the next view and places them in the *pendingDelivery* queue. When all of these messages are ready to be delivered, the process transitions to the *Deliver and Install* state.

In the *Deliver and Install* state, the protocol delivers all of the messages in the *pendingDelivery* queue to the application, followed by a UMC message based on the information in the MC message received

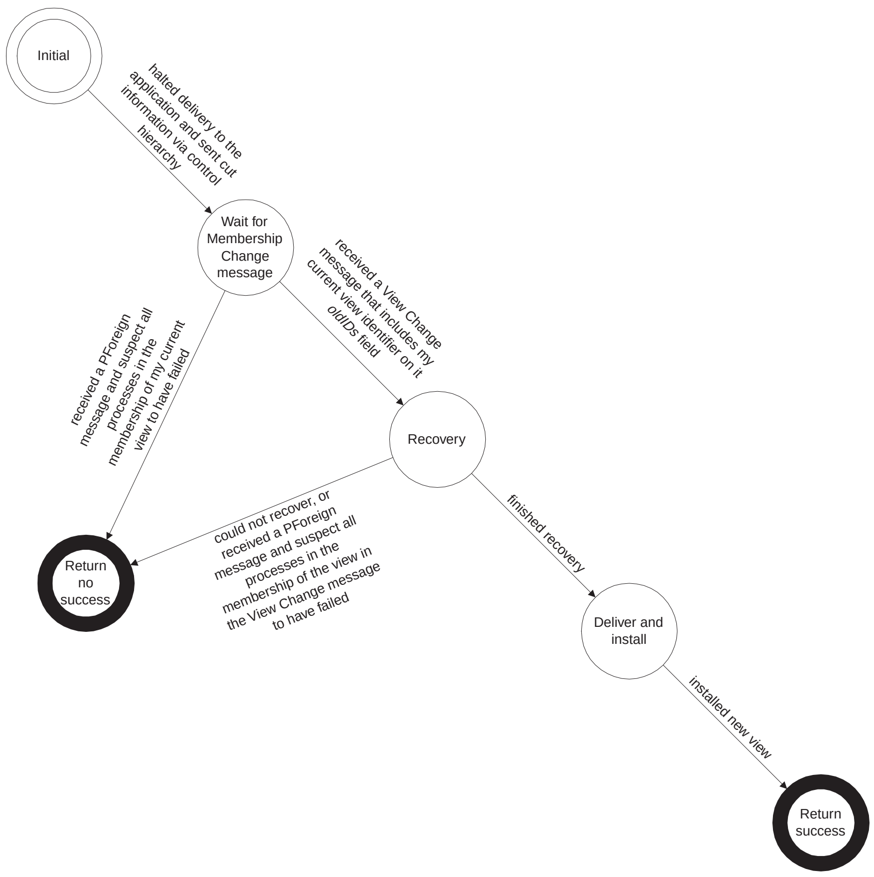
Initial
halted delivery to the application and sent cut information via control hierarchy
Wait for Membership Change message
received a View Change message that includes my current view identifier on it *oldIDs* field
received a PForeign message and suspect all processes in the membership of my current view to have failed
Recovery
could not recover, or received a PForeign message and suspect all processes in the membership of the view in the View Change message to have failed
finished recovery
Return no success
Deliver and install
installed new view
Return success

Figure 6. The Receiver Membership Repair Algorithm.

from the new leader. The next sequence number of the message to be delivered from each process in the membership of the new view is set according to the information in the MC message, if that value is not already larger. Finally, the new view is installed, delivery of messages to the user is continued, and the sending of PLeave messages is allowed again, and the RMRA is deemed successful.

The RMRA is deemed unsuccessful if (1) all of the processes in the membership of this process' current view are suspected of having failed before the MC message is received and a PForeign message is received, (2) all of the processes in the membership of the view specified in the MC are suspected of having failed after the MC is received and a PForeign message is received, or (3) the process is not able to complete the recovery phase.

7. Conclusion and Future Work

One of the principal problems with scaling group communication systems has been the need to maintain a consensus-based membership for the whole group. This membership is a necessary component to providing such properties as group ordering and virtual synchrony.

This paper presents a novel approach to addressing this problem. This approach is based on the fact that the processes in the group are not necessarily equal. We only explicitly track the membership of the processes that are sending application data in the group. This allows us to maintain the group ordering properties our applications require while maintaining the cost of the membership protocols at a reasonable level.

We have presented the membership protocols for this type of groups. For the most part, these membership protocols rely on direct communication within a given subset of the group. The communication of the other information is made through a control hierarchy, which is intended to provide a more scalable way of aggregating and distributing this information.

Some applications may want to know the membership of the entire process group without the restrictions posed by virtual synchrony. There are many ways to gather this information *e.g.,* RTCP[17] and through the control hierarchy in our system. This is a topic of future work for this project.

References

[1] H. Abdel-Wahab, K. Maly, A. Youssef, E. Stoica, C. M. Overstreet, C. Wild, and A. Gupta. The software architecture and interprocess communications of IRI: an internet-based interactive distance learning system. In *Proceedings of IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96)*, Stanford, CA, June 1996.

[2] D. Agarwal, W. E. Johnston, S. Loken, and B. Tierney. Tools for building virtual laboratories. In *Proceedings of Computing in High Energy Physics*, Rio de Janeiro, Brazil, September 1995.

[3] E. S. Al-Shaer, H. Abdel-Wahab, and K. Maly. HiFi: A new monitoring architecture for distributed system management. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 171–178, Austin, TX, June 1999.

[4] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *Proceedings of the 3rd International Workshop on Services in Distributed and Networked Environments (SDNE)*, pages 84–91, Macau, China, June 1996.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of Distributed Algorithms. 6th International Workshop, WDAG '92*, pages 292–312, Berlin, Germany, November 1992.

[6] O. Babaoglu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[7] K. Berket. *The InterGroup Protocols: Scalable Group Communication for the Internet*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, 2000. Also as technical report LBNL-47152 at Lawrence Berkeley National Laboratory.

[8] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987.

[9] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, Santa Barbara, CA, August 1997.

[10] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, Philadelphia, PA, May 1996.

[11] R. Khazan, A. Fekete, and N. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *Proceedings of the 12th International Symposium on Distributed Comupting (DISC)*, pages 258–272, Andros, Greece, September 1998.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.

[14] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[15] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, September 1997.

[16] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[17] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. IETF Request for Comments: 1889, January 1996.